Lab-5：设计NaiveTPU并行加速矩阵乘法

# 1 实验目标

1. 理解矩阵乘法的拆分，掌握控制流程的设计；
2. 理解模块设计思想，掌握自顶向下的设计方法；
3. 学习FPGA设计和实现流程。

# 2 实验环境

1. Vivado 2019.2/Vitis 2019.2；
2. ZYNQ 7020开发板及其配件；
3. ZYNQ上的Linux系统；
4. 本实验过程中使用到的数据文件可在北航盘下载

<https://bhpan.buaa.edu.cn:443/link/05A012DD0705377B461E7907CB93090C>

<https://bhpan.buaa.edu.cn:443/link/41E8F22F0B806AFB0620E8E4C8A04AC9>

1. **注意：本实验指导书中给出的步骤仅为示意步骤作为参考，每人遇到的情况可能有差异，如果遇到问题可根据实际情况进行探索，或向助教寻求帮助。**

# 3 实验要求

1. 撰写实验报告，并**回答4.3 问题中的问题**。实验报告命名：学号+姓名+实验五。（实验报告撰写细节可参考“实验报告撰写格式”）

2) 将实验报告按时交至课程中心作业处。

# 4 实验内容

## 4.1 矩阵乘法模块的设计

### 4.1.1 设计目标

对于FPGA设计而言，最重要的，是确定设计目标，即所设计的模块需要完成什么样的功能，其次才是如何实现这个功能。根据实验需求，在PS侧完成卷积运算img2col的过程，在PL侧完成矩阵乘法运算，并将结果返回给PS侧。因此，PL侧需要完成的目标即，矩阵的乘法运算，且输入矩阵的大小可变。

按照设计，输入矩阵FM和WM分别满足以下特性：

矩阵Feature（或称Input），大小为M×N，其元素均为uint8格式（无符号8位宽整型数据）；矩阵Weight，大小为N×P，其元素均为sint8格式（有符号8位宽整型数据）。输出矩阵Output，大小为M×P，其元素均为sint32格式（有符号32位宽整型数据）。

### 4.1.2 软硬件接口设计

一般而言，PS侧和PL侧分别由软硬件人员各自设计。因此，软硬件接口需要在一开始首先约定好。为简化设计，采用BRAM完成PS侧和PL侧的交互，即，数据和控制指令均通过BRAM传输。在以下的说明中，PS侧和ARM为同义词，PL侧和FPGA为同义词，不再区分。

依据需求，共需要4个BRAM，分别存储输入的feature矩阵，输入的weight矩阵，控制指令，输出的output结果矩阵。将他们分别命名为：BRAM\_FM32b、BRAM\_WM32b、BRAM\_CTRL32b、BRAM\_OUT32b。其中，BRAM\_FM32b用于存储输入的feature数据，BRAM\_WM32b用于存储输入的weight数据，BRAM\_CTRL32b用于存储控制信号，BRAM\_OUT32b用于存储矩阵乘法输出结果。这里，PS侧和PL侧采用AXI4LITE协议进行通信，数据位宽为32bit，使用的BRAM设置为BRAM Controller模式，相应为32bit位宽。

约定交互流程如下：



**图 1 ARM和FPGA交互流程图**

如图 1 ARM和FPGA交互流程图。ARM首先将input数据（即feature）、weight分别存储在BRAM\_FM32b和BRAM\_WM32b中，之后将矩阵参数、指令flag=1分别写入BRAM\_CTRL32b对应地址，等待flag为0。FPGA（即PL侧）循环检测flag信号，当检测到flag为1时，读取矩阵参数，开始执行。FPGA计算完毕将结果写入BRAM\_OUT后，将flag置0，进入等待flag为1的状态。ARM检测到flag为0后，执行后续处理。

经过测试，MLP和LeNet输入矩阵的大小最大如下：

**表格1：MLP与LeNet网络模型输入矩阵最大规模**

|  |  |  |
| --- | --- | --- |
| **\** | **MLP模型** | **LeNet模型** |
| Feature | (1, 784) | (784, 25) |
| Weight | (784, 100) | (400, 120) |

依据表格1：MLP与LeNet网络模型输入矩阵最大规模，我们约定BRAM地址映射如下：

**表格2：BRAM地址映射**

|  |  |  |  |
| --- | --- | --- | --- |
| **起始地址** | **大小（字节）** | **读/写** | **描述** |
| 0x4000\_0000 | 32K | r/w | Feature mem |
| 0x4002\_0000 | 128K | r/w | Weight mem |
| 0x4004\_0000 | 32K | r/w | Output mem |
| 0x4006\_0000 | 4K | r/w | FLAG：0x0-0x04 (arm读写，PL读写)  指令：0x10-0x17 (arm写，PL读) |

输入矩阵的BRAM大小的计算方式为：

Feature矩阵最大为(784, 25)，一个数据为8bit，因此需要BRAM大小为784×25字节，取32KB。Weight矩阵最大为(784, 100)，一个数据为8bit，因此需要BRAM大小为784×100字节，取128KB。

表格 2 的BRAM地址映射中，指令含义如下：

**表格 3 指令 M\*N（input）×N\*P（weight）**

|  |  |
| --- | --- |
| **范围** | **描述** |
| [15:0] | Feature M |
| [31:16] | Weight P |
| [47:32] | Feature/Weight N |
| [63:48] | NULL |

### 4.1.3 矩阵乘法模块设计

确定模块功能后，需要考虑如何实现本模块的功能。由于输入矩阵的大小不确定，本模块采用矩阵分块的方式，将输入矩阵拆分成小矩阵来分别进行计算。

设为矩阵，为矩阵，分块成

，，

其中，，，的列数分别等于，，，的行数，那么

，

其中

为了降低设计难度，我们假设输入矩阵分块时，均分块为同样大小的子矩阵。为了尽可能提高运算效率，子矩阵分块时，分块的大小应尽量大，这样所需要运算的块数变少，可以缩短整体的运算时间。但受限于FPGA资源，如果分块过大，有可能FPGA无法满足一次矩阵运算所消耗的资源数量。因此，需要对器件资源进行评估。

由于本次实验采用ZYNQ 7020开发板，其LUT资源为53200，BRAM资源为140。结合lab3中的MAC模块，通过预综合方式（直接对某部分代码进行综合），得知一个MAC模块大概消耗LUT资源为230，一个Multiply\_8x8模块大概消耗LUT资源为14720（230×64）。其中，Multiply\_8x8模块就是一个8×8大小的矩阵块的运算单元，其中包含64个MAC运算模块。

通过对资源的分析，我们得知，在这块板子上最多可以放下3个Multiply\_8x8模块。但出于便于设计的考虑，同时也是为了预留一些资源给其他控制逻辑，确定使用两个Multiply\_8x8模块同时进行矩阵乘法运算。此时，可以确定矩阵分块的大小为8×16。

前面提到，PS侧和PL侧采用AXI4LITE协议进行通信，数据位宽为32bit，这意味着一个时钟只能传输4个数据（每个数据为8bit）。而对于8×16的子矩阵运算单元（两个Multiply\_8x8模块），每个时钟两个输入接口分别需要输入8个数和16个数。当我们将PS侧传来的输入矩阵存入32bit位宽的BRAM后，如果不进行位宽转换，那么该BRAM在一个时钟下也只能输出32bit，无法满足8×16的子矩阵运算单元的需求，这会降低运算效率。因此，当输入矩阵存入32位宽的BRAM后，还应分别对其进行位宽转换，放入64位宽和128位宽的BRAM当中。

对于每个子矩阵的运算，其本质过程就是将输入搬移到8×16的子矩阵运算单元，再将输出结果搬走。由于矩阵分块有可能引入一些“残余的块”，它们会被填充成相同的大小，因此在搬移输出结果的时候，需要知道有效数据部分的大小。此外，在每次对子矩阵进行运算的时候，都需要知道计算的子矩阵位于原矩阵的哪个部分。还需要知道什么时候可以判定原矩阵的乘法已经计算完毕，即所有子矩阵都已经完成运算。

采用自顶向上的设计思想。首先需要有一个整体的控制模块，在开始运算前，将输入矩阵进行位宽转换。开始运算时，对输入矩阵进行分块，控制参与本次子矩阵运算的是哪两个“块”，并控制将输出结果裁剪为合适的大小，搬移到合适的位置。之后，需要有具体完成位宽转换功能的模块，负责将输入的矩阵格式转换为需要的矩阵格式；需要有具体完成子矩阵相乘的模块，负责将子矩阵数据搬移到运算模块，并控制本次子矩阵运算结果的裁剪；需要有具体完成子矩阵结果拼接的模块，负责依据本次运算子矩阵所处的位置，将结果放在输出BRAM的正确地址。而8×16的子矩阵运算单元，是由两个Multiply\_8x8模块拼接而成的。出于上述考虑，设计矩阵乘法模块结构如图 2 矩阵乘法模块框图所示。



**图 2 矩阵乘法模块框图**

如图 2 矩阵乘法模块框图，CTRL模块是PL侧的总控制模块。初始状态下，等待flag信号为1，该信号从BRAM\_CTRL中读取。检测到flag为1后，分别通知FM\_reshape和WM\_reshape开始工作，并读取计算参数。

FM\_reshape负责将feature数据重构，存储到BRAM\_FM64b中，其数据位宽为64bit。WM\_reshape负责将weight数据重构，存储到BRAM\_WM128b中，其数据位宽为128bit。将数据进行重构而不直接读取原本BRAM中数据进行计算的原因是，BRAM在一个时钟周期内只能访问一个地址的数据，如果读取32bit位宽的BRAM作为Multiply\_8x8的输入，需要连续读取两个时钟以拼接为64bit（一个数据8bit，8个数据64bit），这会影响Multiply\_8x8模块的计算。在对原矩阵进行计算时，是通过将原矩阵拆解为子矩阵进行计算的，feature和weight中的子矩阵会被多次访问。因此，在首次得到数据后，将其进行拼接，主要目的是避免后续多次访问时，每次都要进行拼接操作。

在FM\_reshape和WM\_reshape完成工作后，通知CTRL模块。CTRL模块首先依据计算参数，对原矩阵进行拆解，将子矩阵的数据存储地址送给Multiply\_ctrl模块，通知其开始执行矩阵计算。Multiply\_ctrl模块依据子矩阵数据地址，从BRAM\_FM\_64b和BRAM\_WM\_128b中搬移数据，分别送给两个Multiply\_8x8模块进行子矩阵运算。其中，BRAM\_FM\_64b的数据同时送给两个Multiply\_8x8模块，BRAM\_WM\_128b的数据高低64bit分别送给两个Multiply\_8x8模块。

Multiply\_8x8模块中有8×8个MAC模块，每个MAC模块负责进行向量的乘累加运算（即向量乘）。因此，各MAC模块得出的即为输出矩阵的一个元素。这些输出元素存储在Align\_fifo中，Align\_fifo实际上包括8个fifo，每个fifo存储输出矩阵的一列元素。

Multiply\_ctrl模块将子矩阵相乘运算完成后，反馈给CTRL模块。CTRL模块将该子矩阵结果在输出矩阵中的存储地址送给Out\_ctrl模块。Out\_ctrl模块将输出结果搬移到BRAM\_OUT中。搬移完毕后，通知CTRL模块，由CTRL模块启动下一个子矩阵的运算。

待CTRL模块判定所有子矩阵计算完毕后，置flag信号为0，等待ARM启动下一次的矩阵运算。

### 4.1.4 数据格式约束

依据图 2 矩阵乘法模块框图，需要对矩阵数据在各个BRAM中的存储形式进行约束。**“矩阵乘法模块中的数据形式.xlsx”**中两个矩阵A、B为例说明数据在各个BRAM中的存储形式。设定矩阵大小如下：

A：M×N

B：N×P

输出：M×P

举例：M——10，N——30，P——20

另外，**“NaiveTPU\_硬件矩阵乘法原理.pptx”**中的**数据形式（举例）**部分也可以帮助理解。

下面对子矩阵地址计算进行部分说明。

定义符号“//”：a//b表示a除b的商向下取整。

对于M×N的矩阵，在BRAM\_FM\_64b中的存储，共有[(M-1)//8+1]个子矩阵，相邻子矩阵首地址偏移量为1，对于单个子矩阵，在BRAM\_FM\_64b中，相邻数据地址偏移为[(M-1)//8+1]（子矩阵个数和单个子矩阵内相邻数据地址偏移一致）。

对于N×P的矩阵，在BRAM\_WM\_128中的存储，共有[(P-1)//16+1]个子矩阵，相邻子矩阵首地址偏移量为1，对于单个子矩阵，在BRAM\_WM\_128b中，相邻数据地址偏移为[(P-1)//16+1]（子矩阵个数和单个子矩阵内相邻数据地址偏移一致）。

在计算两个矩阵相乘时，首先将其拆成子矩阵进行计算。规定子矩阵的计算顺序为先横向，再纵向。例如，FM（M×N）有子矩阵FM1和FM2，WM（N×P）有子矩阵WM1和WM2，那么，计算子矩阵的顺序为：FM1×WM1，FM1×WM2，FM2×WM1，FM2×WM2。

对于输出矩阵，已知其大小为M×P，每个子矩阵均不大于8×16的规模。对于每个子矩阵中的行，其相邻数据偏移为1，对于每个子矩阵中的列，其相邻数据偏移为P。对于行方向的子矩阵，其对应位置数据地址偏移相差16；对于列方向的子矩阵，其对应位置数据地址偏移相差8\*P。

例如，“矩阵乘法模块中的数据形式.xlsx”中，输出矩阵子矩阵2数据0c16和0c17在同一行且相邻，在BRAM\_OUT\_32b中的地址差1；子矩阵2数据0c16和1c16在同一列且相邻，地址相差P=20；子矩阵1和子矩阵2为行方向相邻子矩阵，对应位置数据，如0c0和0c16，地址相差16；子矩阵1和子矩阵3为列方向相邻子矩阵，对应位置数据，如0c0和8c0，地址相差8\*P =160。

### 4.1.5 矩阵乘法模块各子模块设计

按照图 2 矩阵乘法模块框图，还需要将各个模块功能进行细化。为降低设计复杂度，主要采用状态机进行控制模块的设计。

#### 4.1.5.1 总控制模块CTRL

根据前面描述的FPGA数据处理流程，CTRL模块状态机分为以下状态：



**图 3 CTRL模块状态转移图**

IDLE：初始状态，循环读取控制BRAM\_CTRL中的FLAG信号，直到该信号为1，跳转到GET\_COMMAND状态。

GET\_COMMAND：读取BRAM\_CTRL中的指令，获取指令参数M、N、P，跳转到INFO\_RESHAPE状态。如果发现指令异常（M、N、P任一为0），则跳转到FINISH状态。该状态拆分成GET\_COMMAND1（获取M、P）和GET\_COMMAND2（获取N）来执行。

INFO\_RESHAPE：通知FM\_reshape和WM\_reshape依据指令参数M、N、P开始工作，跳转到WAIT\_RESHAPE状态。

WAIT\_RESHAPE：等待FM\_reshape和WM\_reshape完成。之后跳转到COM\_SUBADDR状态。

COM\_SUBADDR：依据指令参数M、N、P以及当前子矩阵位置计算下一个子矩阵在BRAM\_FM和BRAM\_WM中的存储地址，并计算该子矩阵在BRAM\_OUT中的存储地址。跳转到INFO\_MULTIPLY状态。

INFO\_MULTIPLY：通知Multiply\_ctrl模块按照子矩阵在BRAM\_FM和BRAM\_WM中的存储地址开始工作，跳转到WAIT\_MULTIPLY状态。

WAIT\_MULTIPLY：等待Multiply\_ctrl模块完成工作。之后跳转到INFO\_OUT状态。

INFO\_OUT：通知Out\_ctrl模块按照子矩阵在BRAM\_OUT中的存储地址开始工作，跳转到WAIT\_OUT。

WAIT\_OUT：等待Out\_ctrl模块完成工作。之后跳转到JUDGE\_ FINISH状态。

JUDGE\_FINISH：判断当前子矩阵是否为最后一个子矩阵。若是，则跳转到FINISH状态；若不是，则跳转到COM\_SUBADDR状态。

FINISH：修改FLAG标记，跳转到IDLE状态。

#### 4.1.5.2 FM数据重组模块FM\_reshape

IDLE：初始状态，等待CTRL模块通知开始工作。之后，跳转到COM状态。

COM：依据输入的M、N计算需要执行的次数。其中，M是BRAM\_FM\_32b中每次拼接前数据的个数，M（＞0）个数需要在BRAM\_FM\_32b中存储[(M-1)//4+1]行。在BRAM\_FM\_64b中，则需要存储[(M-1)//8+1]行。之后，跳转到WORK状态。

WORK：执行搬移操作。以[(M-1)//4+1]为第一级循环，N为第二级循环，将BRAM\_FM\_32b中的数据通过拼接搬移到BRAM\_FM\_64b中。在拼接时，若不足一行的拼接，需要补0。如果遇到FM矩阵新行，此时无需进行拼接，补0即可。完成搬移后，跳转到FINISH状态。

FINISH：反馈给CTRL模块，告知已完成搬移。之后，跳转到IDLE状态。

#### 4.1.5.2 WM数据重组模块WM\_reshape

IDLE：初始状态，等待CTRL模块通知开始工作。之后，跳转到COM状态。

COM：依据输入的P、N计算需要执行的次数。其中，P是BRAM\_WM\_32b中每次拼接前数据的个数，P（＞0）个数需要在BRAM\_WM\_32b中存储[(P-1)//4+1]行。在BRAM\_WM\_128b中，则需要存储[(P-1)//16+1]行。之后，跳转到WORK状态。

WORK：执行搬移操作。以[(P-1)//4+1]为第一级循环，N为第二级循环，将BRAM\_WM\_32b中的数据通过拼接搬移到BRAM\_WM\_128b中。在拼接时，若不足一行的拼接，需要补0。如果遇到WM矩阵新行，此时无需进行拼接，补0即可。完成搬移后，跳转到FINISH状态。

FINISH：反馈给CTRL模块，告知已完成搬移。之后，跳转到IDLE状态。

#### 4.1.5.3 子矩阵相乘控制模块Multiply\_ctrl

IDLE：初始状态，等待CTRL模块通知开始工作。之后，跳转到INFO状态。

INFO：告知Align\_fifo本次运算子矩阵的规模（从CTRL模块获取），以免Align\_fifo将多余的0写入。之后，跳转到WORK状态。

WORK：依据从CTRL模块获取的数据个数N、子矩阵首数据地址、子矩阵地址递增量（[(M-1)//8+1]和[(P-1)//16+1]），执行子矩阵相乘的逻辑控制。其本质上，是将数据搬移送入Multiply\_8x8模块。完成搬移后，跳转到WAIT状态。

WAIT：等待Align\_fifo模块获取子矩阵相乘结果。当Align\_fifo模块获取结果后，产生align\_fifo\_get\_all信号通知本模块。两个Align\_fifo模块均获取运算结果后，跳转到FINISH状态。

FINISH：反馈给CTRL模块，告知已完成子矩阵相乘运算。之后，跳转到IDLE状态。

#### 4.1.5.4 子矩阵输出缓存模块Align\_fifo

依据Multiply\_ctrl模块告知的子矩阵运算结果规模，将输出数据分别存储在8个FIFO当中。在接收数据时，需要对数据进行判断是否为所需要的数据，因为Multiply\_8x8模块输出的总是8×8规模的子矩阵，其中包含了一些不需要的0数据。由于需要的子矩阵结果总是存在于8×8子矩阵的左上角，不需要的填充0数据均在右侧和下侧，因此依据本次子矩阵结果规模，以及一些计数器，就可以判定哪些数据是实际需要的。

当收集到足够数据后，将align\_fifo\_get\_all信号置1，告知Multiply\_ctrl模块。之后等待Out\_ctrl模块控制输出（通过out\_ctrl\_ready信号）。输出时，依次轮询各个FIFO，但不能超过本次子矩阵的大小。

#### 4.1.5.5 输出控制模块Out\_ctrl

IDLE：初始状态，等待CTRL模块通知开始工作。之后，跳转到WORK状态。

WORK：依据从CTRL模块获取的子矩阵首数据坐标（Ma、Pa），结果矩阵规模（P），子矩阵结果大小（sub\_P，sub\_M），将输出子矩阵结果搬移到BRAM\_OUT\_32b中。之后，跳转到FINISH状态。

FINISH：反馈给CTRL模块，告知已完成结果搬移。之后，跳转到IDLE状态。

#### 4.1.5.6 Multiply\_8x8模块结构



**图 4 Multiply\_8x8模块框图**

#### 4.1.5.7 MAC模块结构

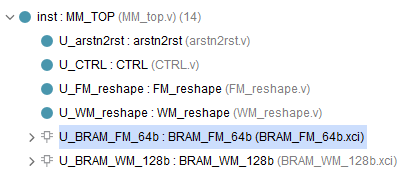
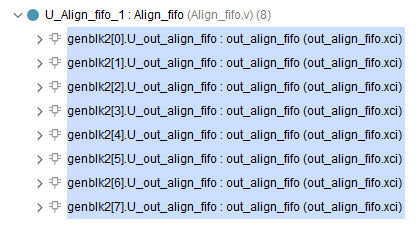


**图 5 MAC模块框图**

## 4.2 矩阵乘法模块的实现

### 4.2.1 IP核的使用

IP核，即知识产权核，是“用于ASIC或FPGA中的预先设计好的电路功能模块”。调用IP核能避免重复劳动，大大减轻工程师的负担。在矩阵乘法模块的设计中，主要用到FIFO和BRAM的IP核。



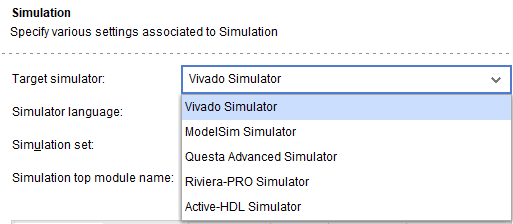
**图 6 矩阵乘法模块中使用到的IP核（左图：FIFO，右图：BRAM）**

关于它们的使用，网上很容易找到介绍，此处不再赘述。

### 4.2.2 仿真

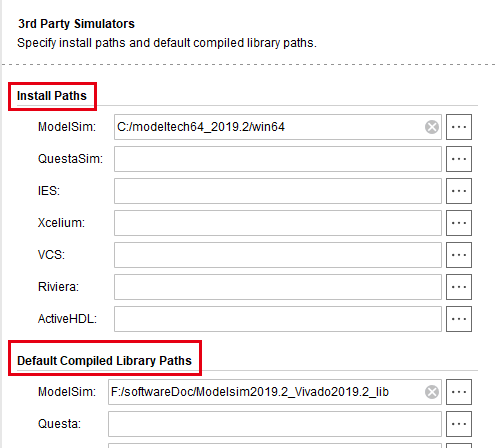
#### 4.2.2.1 仿真器的选择

在lab3中，使用了Vivado自带的仿真器。在菜单栏->Tools->Settings->Simulation->Target simulator中可以选择其他仿真器。



**图 7 Vivado中仿真器的选择**

使用其他仿真器，同时需要在菜单栏->Tools->Settings->3rd Party Simulators中选择安装目录以及对应的编译库路径。

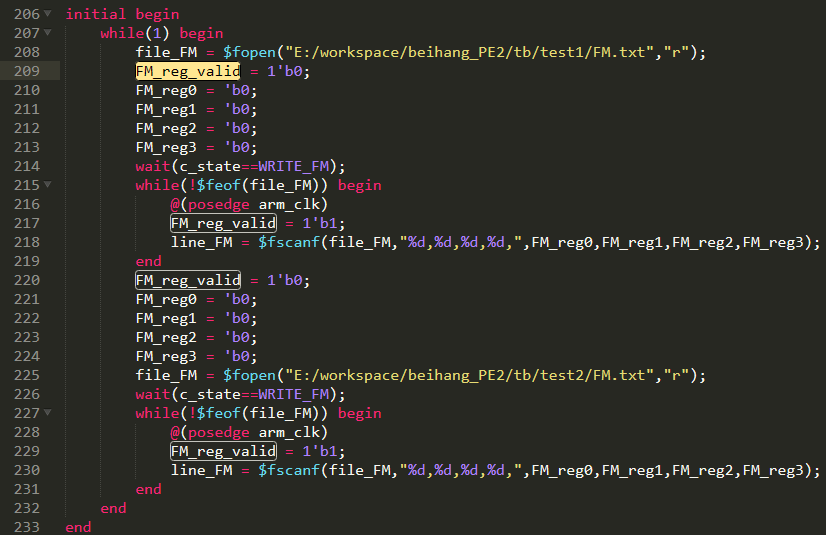


**图 8 Vivado中第三方仿真器设置**

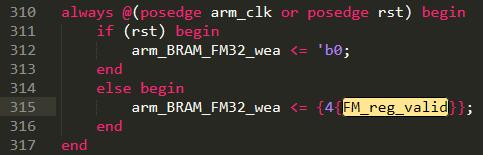
第三方仿真器不是必须的。如对modelsim仿真器有兴趣，可以搜索“vivado modelsim联合仿真”词条。

但是**需要特别注意的是，不同的仿真器对同样的代码可能有不同的解释，尤其是阻塞赋值和非阻塞赋值**。因此，在写激励文件的时候，需要特别注意initial块中的“=”和“<=”，它们在不同的仿真器中可能存在不同的解释，往往表现为仿真出来的信号存在一拍的偏差。

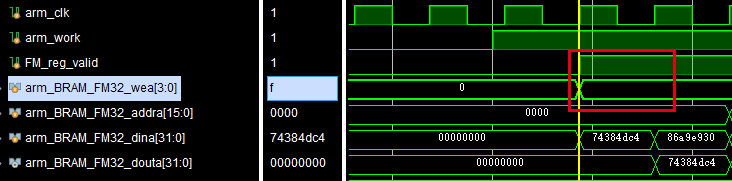
例如，在initial块中使用“=”时，注意图中的FM\_reg\_valid信号：



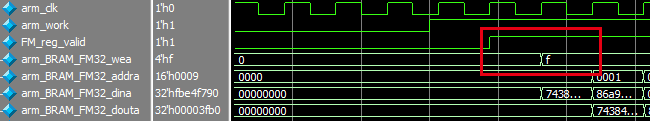
**图 9 FM\_reg\_valid在initial中使用“=”**



**图 10 将FM\_reg\_valid打拍**



**图 11 Vivado仿真器结果**



**图 12 modelsim仿真结果**

可以看到，两个仿真器对阻塞赋值“=”的解释并不一致，尽管FM\_reg\_valid信号波形看起来是一致的，但对下一级的寄存器arm\_BRAM\_FM32\_wea确实产生了影响。

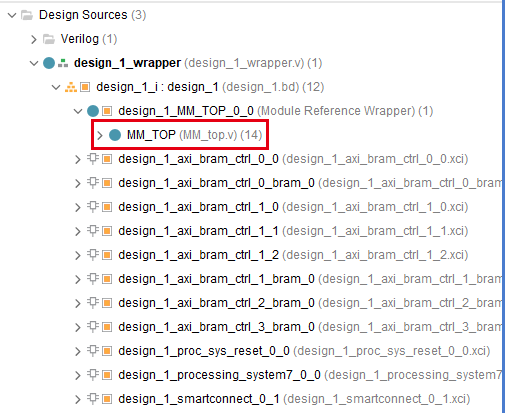
鉴于两种仿真器的不同行为，在代码中做了一些区分，使用参数SIMULATOR进行选择。



**图 13 参数SIMULATOR**

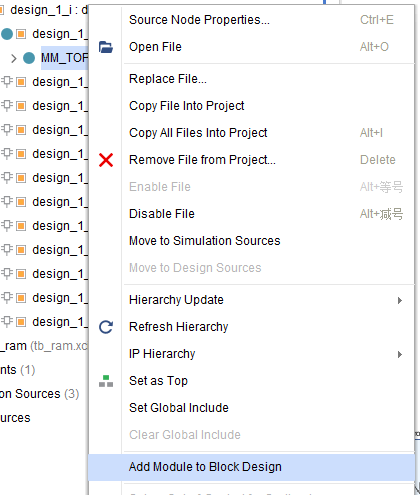
#### 4.2.2.2 工程结构

本次实验所需要的工程已提供，可以直接使用。本小节下述部分为操作提示，本工程已经完成，不需要再次执行。

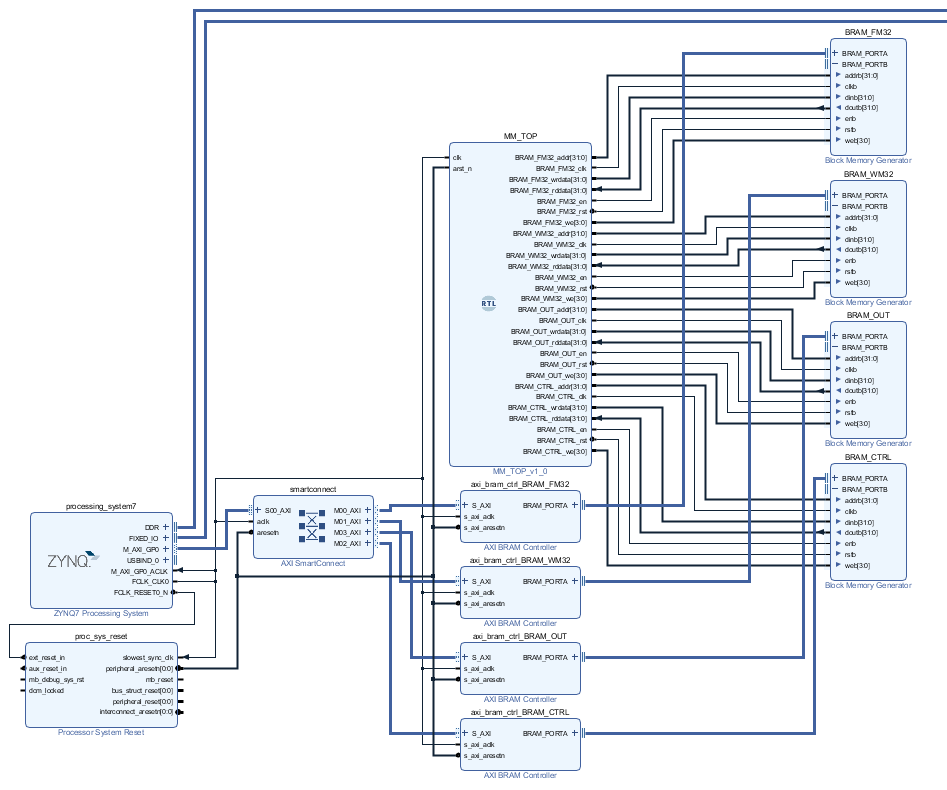


**图 14 工程结构**

如图 14 工程结构，MM\_TOP模块是矩阵乘法模块，其它模块均是在Block Design中添加的。在Block Design中，右键MM\_TOP模块，选择Add Module to Block Design就可将自己设计的模块添加到其中。

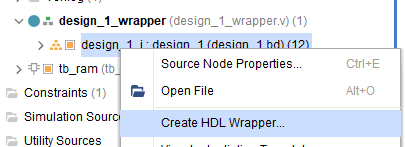


**图 15 选择Add Module to Block Design（本工程中已经添加）**



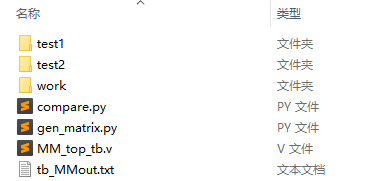
**图 16 Block Design连线图**

顶层的design\_1\_wrapper文件可以由Vivado自动生成。其中，design\_1文件即创建的Block Design文件。



**图 17 Vivado自动生成顶层文件（本工程中已经添加）**

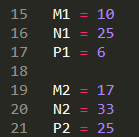
#### 4.2.2.3 仿真环境



**图 18 工程目录下tb文件夹**

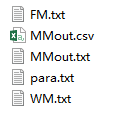
如图 18 工程目录下tb文件夹，test1和test2文件夹是输入矩阵，由gen\_matrix.py生成。Work文件夹是文本编辑器产生的临时文件，不需要关注，可以删除。Compare.py用于对比输入文件和仿真结果是否一致。Gen\_matrix.py用于产生随机输入矩阵。MM\_top\_tb.v是本模块的激励文件，该文件模拟了ARM的行为，将输入送入矩阵乘法模块，并将返回的结果写入tb\_MMout.txt文件中。

打开gen\_matrix.py文件，根据需要修改参数如下：



**图 19 gen\_matrix.py需要设定参数**

其中，M1、N1、P1是输入1的参数，生成的文件保存在test1下；M2、N2、P2是输入2的参数，生成的文件保存在test2下。对于每个输入，依据M、N、P产生元素值随机的FM矩阵和WM矩阵，并计算他们的矩阵乘结果，保存下来。



**图 20 gen\_matrix.py生成文件**

打开MM\_top\_tb.v，**修改所有绝对路径为自己工程目录**。



**图 21 类似路径有多处，注意全部修改**

**依据自己的仿真器，修改参数**。



**图 22 修改仿真器参数**

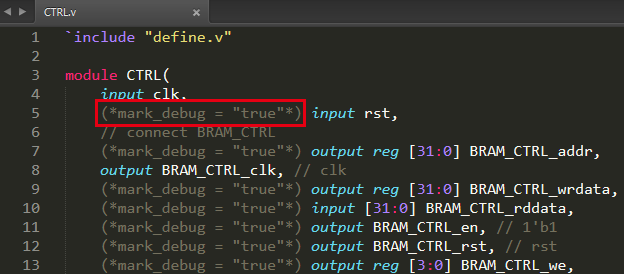
之后，在Vivado中执行仿真。仿真激励会循环将test1和test2送入矩阵乘法模块，并将矩阵乘法模块的输出写入tb\_MMout.txt。

停止仿真后，执行compare.py。如出现数据结果不匹配的情况，该脚本会将其打印。如提示“data right”，说明数据结果匹配，仿真正确。

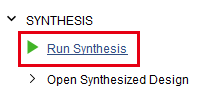
### 4.2.3 调试和上板

#### 4.2.3.1 抓信号

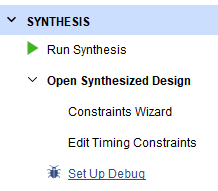
Vivado提供抓取信号波形的功能，可以实现在线调试。该功能主要通过ila的IP核实现。下面介绍一种方法。



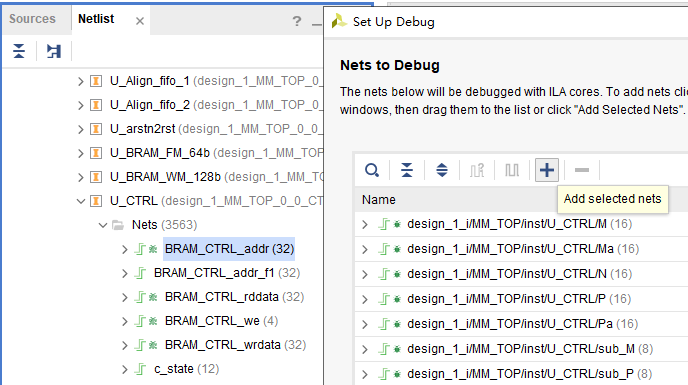
**图 23 在需要抓取的信号前添加(\*mark\_debug = "true"\*)，保存文件**



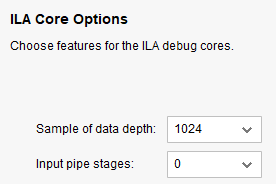
**图 24 点击综合，等待综合完成**



**图 25 点击Set Up Debug**

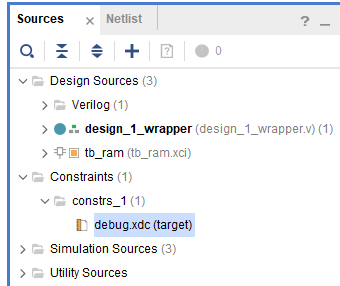


**图 26 在Netlist中找到相应信号，点击Add selected nets进行添加**



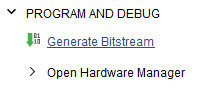
**图 27 设置采样深度和流水线级数，可不修改**

**点击Ctrl+S保存综合修改**。

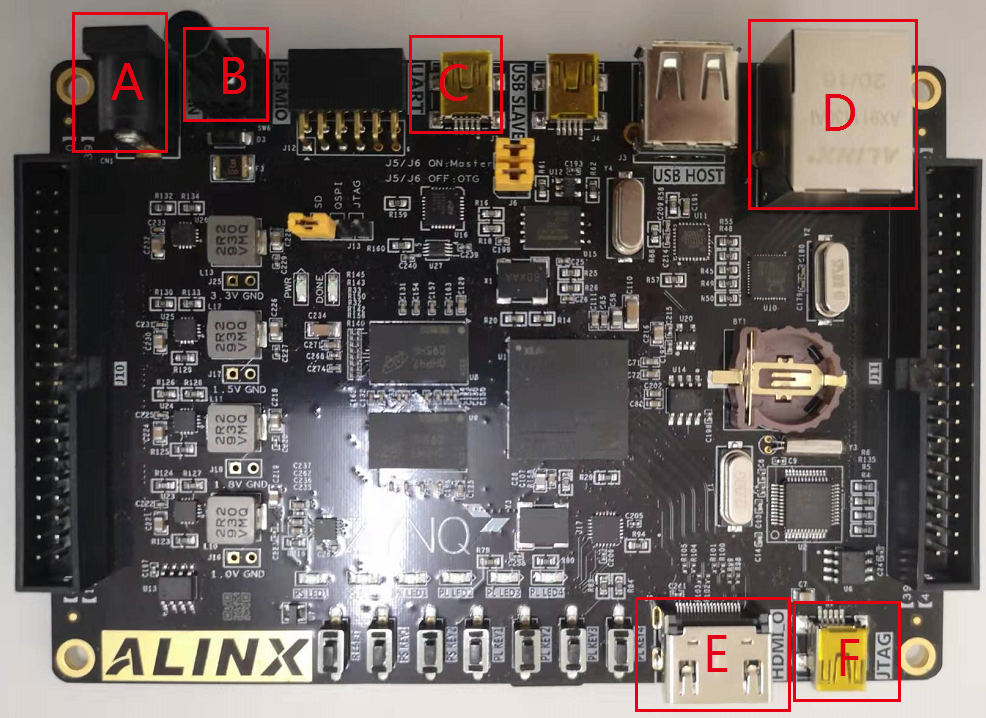


**图 28 打开debug.xdc可以看到添加的ila核**

#### 4.2.3.2 上板

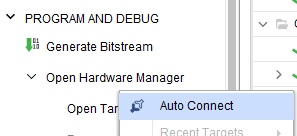


**图 29 直接生成bit文件（会提示首先进行Implementation）**

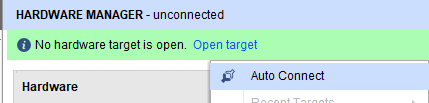


**图 30 A（电源口）；B（板子开关）；C（UART口，用于串口通信）；D（网口，用于网口通信）；E（HDMI口，用于外接显示器）；F（JTAG口，用于下载bit文件和在线调试）**

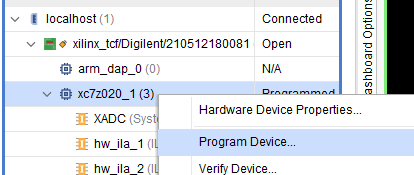
板子和PC进行连线。



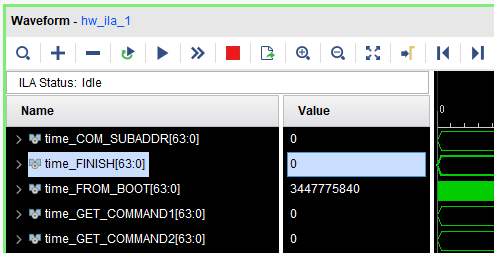
**图 31 连接器件**



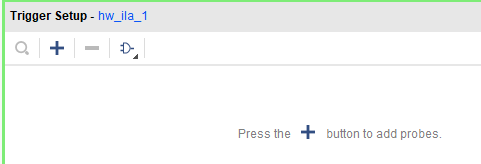
**图 32 连接器件**



**图 33 下载bit流文件**



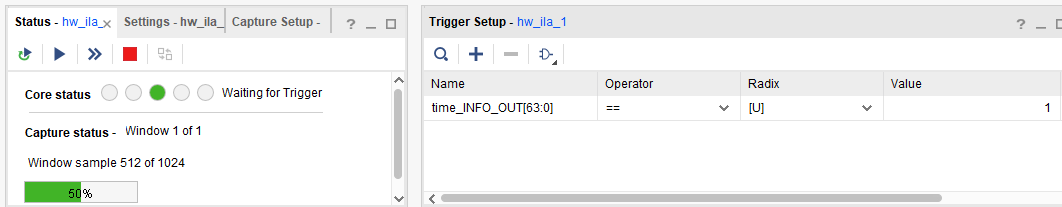
**图 34 在Waveform中看到实时数据**



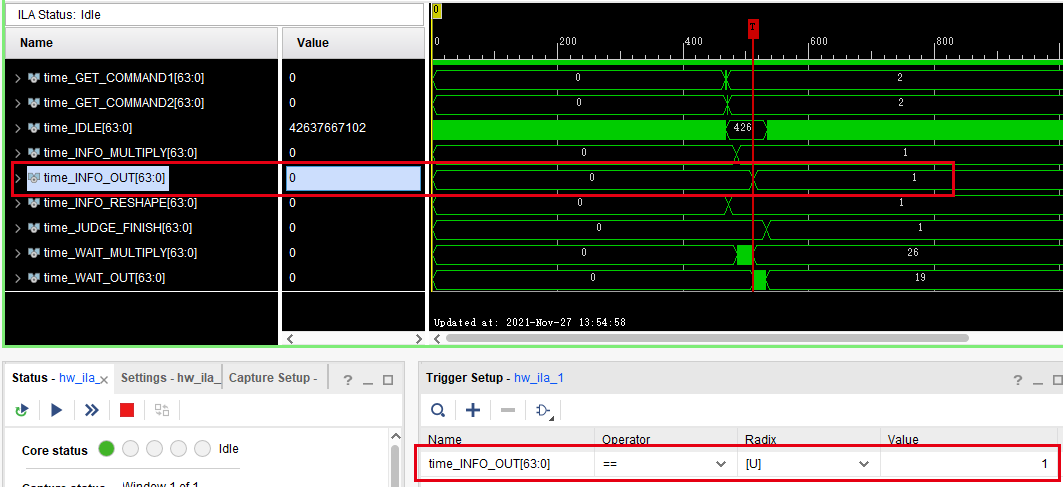
**图 35 在Trigger Setup中设置触发条件**



**图 36 开始触发**

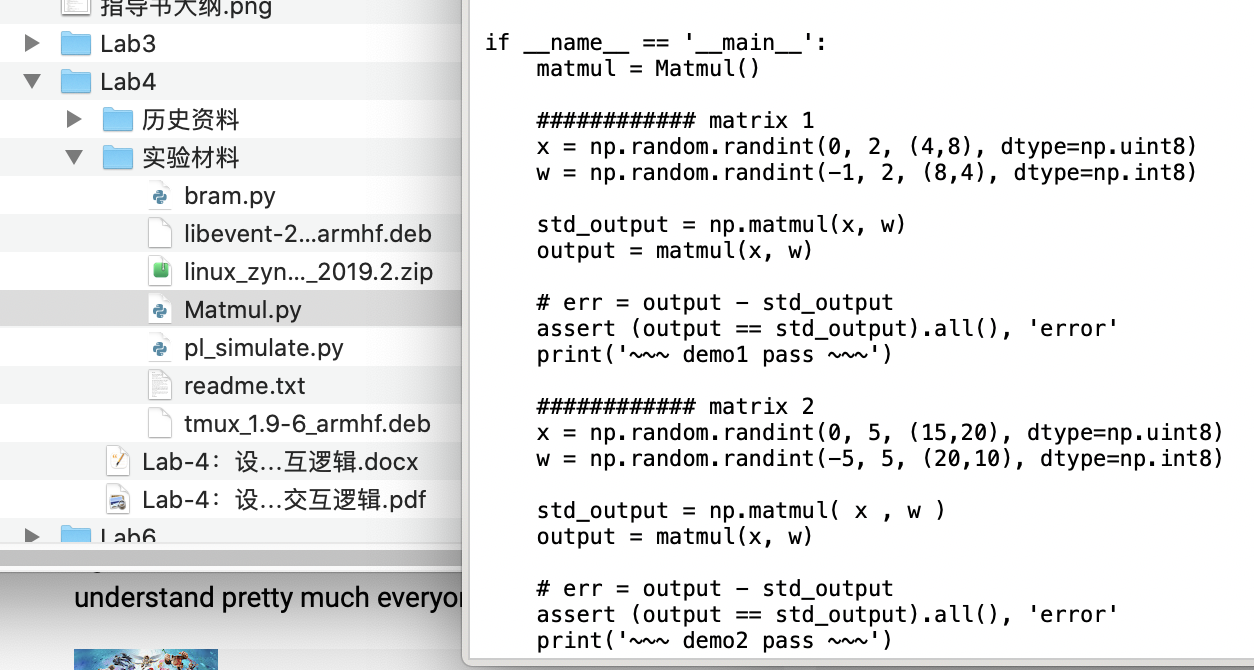


**图 37 等待条件被触发到**

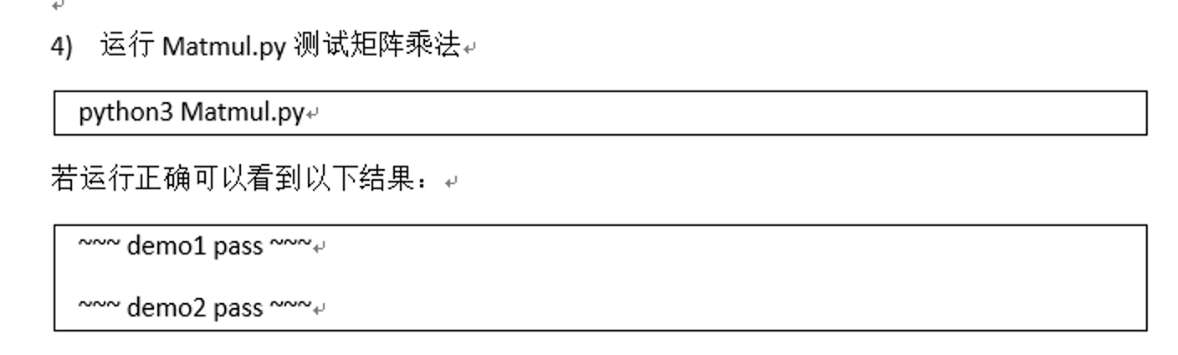


**图 38 PS送入数据后，在设定条件位置处触发**

Lab4中Matmul.py完成了矩阵输入写入BRAM，用pl\_simulate.py模拟了FPGA的操作。将bit下载后，无需执行pl\_simulate.py，执行Matmul.py即可。



**图 39 lab4代码**



**图 40 应打印结果**

## 4.3 问题

本部分问题无需全部完成，但至少完成**30※**。完成※越多，本次实验得分越高。

### 4.3.1 问题1（3※）

依据工程中BRAM\_FM64和BRAM\_WM128的位宽、地址深度，写出M、N、P取值的限制。

### 4.3.2 问题2（3※）

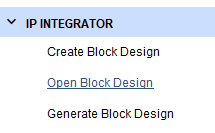
依据mlp和lenet模型，判断当前BRAM\_FM32、BRAM\_WM32、BRAM\_FM64和BRAM\_WM128大小是否足够。

### 4.3.3 问题3（10※）

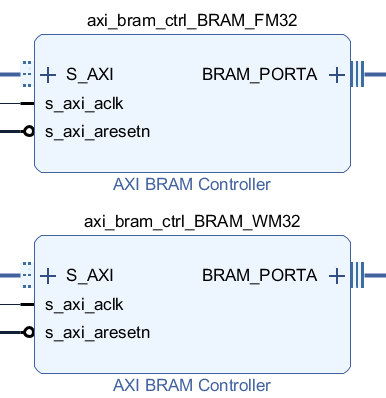
修改MM\_top\_tb.v，利用$random函数在MM\_top\_tb.v中产生随机的M、N、P，依据产生的M、N、P来产生随机输入矩阵。之后，计算两个随机矩阵相乘的结果（三层for循环嵌套，注意不需要有时钟限制，仿真表现为立马算出结果），将结果暂存。将随机矩阵送入MM\_top模块，待得到运算结果，和暂存的结果进行比较，打印出是否出错等必要信息。重复以上过程，不停地产生随机输入进行计算和结果比对，对MM\_top模块进行充分的、随机化的验证。

### 4.3.4 问题4（8※）

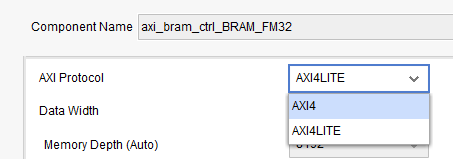
如图 2 矩阵乘法模块框图，输入矩阵首先写入BRAM\_FM\_32b和BRAM\_WM\_32b，再转换写入BRAM\_FM\_64b和BRAM\_WM\_128b。修改代码和工程，将PS侧输入矩阵直接写进BRAM\_FM\_64b和BRAM\_WM\_128b。提示，PS侧和PL侧的交互不一定非要采用AXI4LITE协议，这意味着总线的数据位宽不一定非得是32bit。



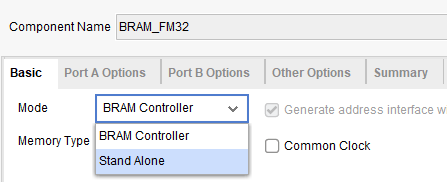
**图 41 打开Block Design**



**图 42 双击AXI BRAM Controller进行设置**



**图 43选择AXI协议**



**图 44设置BRAM模式**

**（可以在Stand Alone模式下修改位宽后切换回BRAM Controller模式）**

**注意，输入矩阵直接写进BRAM\_FM\_64b和BRAM\_WM\_128b时，PS侧补0的个数也需要进行修改。本题目另提供工程，无须对原有工程进行修改，修改PS侧补0代码，跑通流程即可。**

### 4.3.5 问题5（3※）

PS侧输入矩阵会进行补0操作。从PL操作流程考虑，补0是否必要？如果不进行补0，无效位置保留默认值，是否会影响矩阵运算结果？给出原因。

### 4.3.6 问题6（8※）

Lab4中实现了对input、weight补零后写入BRAM的Matmul类，补零操作存在一定时间开销。修改Matmul类实现，通过跳地址的方式将input、weight写入对应的BRAM块中。注意，有效数据存储位置相同，但跳地址的方式不进行补零，跳过本来应该填充零的位置。

### 4.3.7 问题7（2※）

本工程中FPGA的时钟频率是多少？提示，在Block Design中双击ZYNQ7 Processing System，点击Clock Configuration查看PL Fabric Clocks的时钟。

### 4.3.8 问题8（5※）

仿照图 3 CTRL模块状态转移图，画出Multiply\_ctrl模块的状态转移图。并给出以下寄存器的功能：sub\_scale\_M1、sub\_scale\_P1、sub\_scale\_M2、sub\_scale\_P2。

### 4.3.9 问题9（8※）

阅读Align\_fifo模块代码，结合仿真波形描述数据写入和读出的过程。

### 4.3.10 问题10（n※）

除了上述问题提到的，提出你认为设计上还存在改进的点，或提出你的完成本模块功能的方案。本题不设※上限，合理并有效的改进点计5※，提出创新可行的方案思路计12※，给出详细的方案计25※。